Efficient Seeds Computation Revisited

Michalis Christou, Maxime Crochemore, Costas S. Iliopoulos, Marcin Kubica, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, Bartosz Szreder, Tomasz Walen

Efficient Seeds Computation Revisited*

Michalis Christou[1], Maxime Crochemore[1,3], Costas S. Iliopoulos[1,4], Marcin Kubica[2], Solon P. Pissis[1], Jakub Radoszewski[2], Wojciech Rytter[2,5***], Bartosz Szreder[2], and Tomasz Waleń[2]**

[1] Dept. of Informatics, King's College London, London WC2R 2LS, UK
[michalis.christou,maxime.crochemore,csi,solon.pissis]@dcs.kcl.ac.uk
[2] Dept. of Mathematics, Computer Science and Mechanics,
University of Warsaw, Warsaw, Poland
[kubica,jrad,rytter,szreder,walen]@mimuw.edu.pl
[3] Université Paris-Est, France
[4] Digital Ecosystems & Business Intelligence Institute,
Curtin University of Technology, Perth WA 6845, Australia
[5] Dept. of Math. and Informatics,
Copernicus University, Toruń, Poland

Abstract. The notion of the cover is a generalization of a period of a string, and there are linear time algorithms for finding the shortest cover. The seed is a more complicated generalization of periodicity, it is a cover of a superstring of a given string, and the shortest seed problem is of much higher algorithmic difficulty. The problem is not well understood, no linear time algorithm is known. In the paper we give linear time algorithms for some of its versions — computing shortest left-seed array, longest left-seed array and checking for seeds of a given length. The algorithm for the last problem is used to compute the seed array of a string (i.e., the shortest seeds for all the prefixes of the string) in $O(n^2)$ time. We describe also a simpler alternative algorithm computing efficiently the shortest seeds. As a by-product we obtain an $O(n \log(n/m))$ time algorithm checking if the shortest seed has length at least m and finding the corresponding seed. We also correct some important details missing in the previously known shortest-seed algorithm (Iliopoulos et al., 1996).

1 Introduction

The notion of periodicity in strings is widely used in many fields, such as combinatorics on words, pattern matching, data compression and automata theory (see [13, 14]). It is of paramount importance in several applications, not to talk about its theoretical aspects. The concept of quasiperiodicity is a generalization of the notion of periodicity, and was defined by Apostolico and Ehrenfeucht in [2]. In a periodic repetition the occurrences of the period do not overlap. In contrast, the quasiperiods of a quasiperiodic string may overlap.

* The authors thank an anonymous referee for proposing several insightful remarks.
** The author is supported by grant no. N206 568540 of the National Science Centre.
*** The author is supported by grant no. N206 566740 of the National Science Centre.

We consider *words* (*strings*) over a finite alphabet Σ, $u \in \Sigma^*$; the empty word is denoted by ε; the positions in u are numbered from 1 to $|u|$. By Σ^n we denote the set of words of length n. By u^R we denote the reverse of the string u. For $u = u_1u_2 \dots u_n$, let us denote by $u[i\,..\,j]$ a *factor* of u equal to $u_i \dots u_j$ (in particular $u[i] = u[i\,..\,i]$). Words $u[1\,..\,i]$ are called *prefixes* of u, and words $u[i\,..\,n]$ are called *suffixes* of u. Words that are both prefixes and suffixes of u are called *borders* of u. By $\mathsf{border}(u)$ we denote the length of the longest border of u that is shorter than u. We say that a positive integer p is the (shortest) *period* of a word $u = u_1 \dots u_n$ (notation: $p = \mathsf{per}(u)$) if p is the smallest positive number, such that $u_i = u_{i+p}$, for $i = 1, \dots, n-p$. It is a known fact [6, 8] that, for any string u, $\mathsf{per}(u) + \mathsf{border}(u) = |u|$.

We say that a string s *covers* the string u if every letter of u is contained in some occurrence of s as a factor of u. Then s is called a *cover* of u. We say that a string s is: a *seed* of u if s is a factor of u and u is a factor of some string w covered by s; a *left seed* of u if s is both a prefix and a seed of u; a *right seed* of u if s is both a suffix and a seed of u (equivalently, s^R is a left seed of u^R). Seeds were first defined and studied by Iliopoulos, Moore and Park [11], who gave an $O(n \log n)$ time algorithm computing all the seeds of a given string $u \in \Sigma^n$, in particular, the shortest seed of u.

By $\mathsf{cover}(u)$, $\mathsf{seed}(u)$, $\mathsf{lseed}(u)$ and $\mathsf{rseed}(u)$ we denote the length of the shortest: cover, seed, left seed and right seed of u, respectively. By $\mathsf{covermax}(u)$ and $\mathsf{lseedmax}(u)$ we denote the length of the longest cover and the longest left seed of u that is shorter than u, or 0 if none.

For a string $u \in \Sigma^n$, we define its: *period array* $\mathsf{P}[1\,..\,n]$, *border array* $\mathsf{B}[1\,..\,n]$, *suffix period array* $\mathsf{P}'[1\,..\,n]$, *cover array* $\mathsf{C}[1\,..\,n]$, *longest cover array* $\mathsf{C}^M[1\,..\,n]$, *seed array* $\mathsf{Seed}[1\,..\,n]$, *left-seed array* $\mathsf{LSeed}[1\,..\,n]$, and *longest left-seed array* $\mathsf{LSeed}^M[1\,..\,n]$ as follows:

$$\begin{aligned}
\mathsf{P}[i] &= \mathsf{per}(u[1\,..\,i]), & \mathsf{B}[i] &= \mathsf{border}(u[1\,..\,i]),\\
\mathsf{P}'[i] &= \mathsf{per}(u[i\,..\,n]), & \mathsf{C}[i] &= \mathsf{cover}(u[1\,..\,i]),\\
\mathsf{C}^M[i] &= \mathsf{covermax}(u[1\,..\,i]), & \mathsf{Seed}[i] &= \mathsf{seed}(u[1\,..\,i]),\\
\mathsf{LSeed}[i] &= \mathsf{lseed}(u[1\,..\,i]), & \mathsf{LSeed}^M[i] &= \mathsf{lseedmax}(u[1\,..\,i]).
\end{aligned}$$

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$u[i]$	a	b	a	a	b	a	a	a	b	b	a	a	b	a	a	b
$\mathsf{P}[i]$	1	2	2	3	3	3	3	7	7	10	10	11	11	11	11	11
$\mathsf{B}[i]$	0	0	1	1	2	3	4	1	2	0	1	1	2	3	4	5
$\mathsf{C}[i]$	1	2	3	4	5	3	4	8	9	10	11	12	13	14	15	16
$\mathsf{C}^M[i]$	0	0	0	0	0	3	4	0	0	0	0	0	0	0	0	0
$\mathsf{LSeed}[i]$	1	2	2	3	3	3	3	4	4	10	10	11	11	11	11	11
$\mathsf{LSeed}^M[i]$	0	0	2	3	4	5	6	7	8	0	10	11	12	13	14	15
$\mathsf{Seed}[i]$	1	2	2	3	3	3	3	4	4	8	8	8	8	8	8	11

Table 1. An example string together with its periodic and quasiperiodic arrays. Note that the left-seed array and the seed array are non-decreasing.

The border array, suffix border array and period array can be computed in $O(n)$ time [6, 8]. Apostolico and Breslauer [1, 4] gave an on-line $O(n)$ time algorithm computing the cover array $\mathsf{C}[1\,..\,n]$ of a string. Li and Smyth [12] provided an algorithm, having the same characteristics, for computing the longest cover array $\mathsf{C}^M[1\,..\,n]$ of a given string. Note that the array C^M enables computing all covers of all prefixes of the string, same property holds for the border array B. Unfortunately, the LSeed^M array does not share this property.

Table 1 shows the above defined arrays for $u = \texttt{abaabaaabbaabaab}$. For example, for the prefix $u[1\,..\,13]$ the period equals 11, the border is `ab`, the cover is `abaabaaabbaab`, the left seed is `abaabaaabba`, the longest left seed is `abaabaaabbaa`, and the seed is `baabaaab`.

We list here several useful (though obvious) properties of covers and seeds.

Observation 1
(a) A cover of a cover of u is also a cover of u.
(b) A cover of a left (right) seed of u is also a left (right) seed of u.
(c) A cover of a seed of u is also a seed of u.
(d) If u is a factor of v then $\mathsf{seed}(u) \leq \mathsf{seed}(v)$.
(e) If u is a prefix of v then $\mathsf{lseed}(u) \leq \mathsf{lseed}(v)$.
(f) If s and s' are two covers of a string u, $|s'| < |s|$, then s' is a cover of s.
(g) If s is the shortest cover or the shortest left seed or the shortest seed of a string u then $\mathsf{per}(s) > |s|/2$.

For a set X of positive integers, let us define the *maxgap* of X as:

$$\mathsf{maxgap}(X) \;=\; \max\{b-a : a, b \text{ are consecutive numbers in } X\} \;\text{ or } 0 \text{ if } |X| \leq 1.$$

For example $\mathsf{maxgap}(\{1, 3, 8, 13, 17\}) = 5$.

For a factor v of u, let us define $Occ(v, u)$ as the set of starting positions of all occurrences of v in u. By $first(v)$ and $last(v)$ we denote $\min Occ(v, u)$ and $\max Occ(v, u)$ respectively. For the sake of simplicity, we will abuse the notation, and denote $\mathsf{maxgap}(v) = \mathsf{maxgap}(Occ(v, u))$.

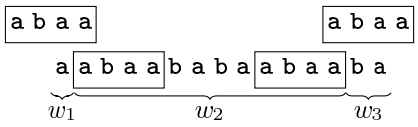

Fig. 1. The word $s = \texttt{abaa}$ is a border seed of $u = \texttt{aabaababaabaaba}$.

Assume s is a factor of u. Let us decompose the word u into $w_1w_2w_3$, where w_2 is the longest factor of u for which s is a border, i.e., $w_2 = u[first(s)\,..\,(last(s) + |s| - 1)]$. Then we say that s is a *border seed* of u if s is a seed of $w_1 \cdot s \cdot w_3$, see Fig. 1. The following fact is a corollary of Lemma 4, proved in Section 2.

Fact 2 *Let s be a factor of $u \in \Sigma^*$. The word s is a border seed of u if and only if* $|s| \geq \max(P[first(s) + |s| - 1],\ P'[last(s)])$.

Notions of maxgaps and border seeds provide a useful characterization of seeds.

Observation 3 *Let s be a factor of $u \in \Sigma^*$. The word s is a seed of u if and only if $|s| \geq \mathsf{maxgap}(s)$ and s is a border seed of u.*

Several new and efficient algorithms related to seeds in strings are presented in this paper. Linear time algorithms computing left-seed array and longest left-seed array are given in Section 2. In Section 3 we show a linear time algorithm finding seed-of-a-given-length and apply it to computing the seed array of a string in $O(n^2)$ time. Finally, in Section 4 we describe an alternative simple $O(n \log n)$ time computation of the shortest seed, from which we obtain an $O(n \log (n/m))$ time algorithm checking if the shortest seed has length at least m (described in Section 5).

2 Computing Left-Seed Arrays

In this section we show two $O(n)$ time algorithms for computing the left-seed array and an $O(n)$ time algorithm for computing the longest left-seed array of a given string $u \in \Sigma^n$. We start by a simple characterization of the length of the shortest left seed of the whole string u — see Lemma 5. In its proof we utilize the following auxiliary lemma which shows a correspondence between the shortest left seed of u and shortest covers of all prefixes of u.

Lemma 4. *Let s be a prefix of u, and let j be the length of the longest prefix of u covered by s. Then s is a left seed of u if and only if $j \geq \mathsf{per}(u)$.*

In particular, the shortest left seed s of u is the shortest cover of the corresponding prefix $u[1\,..\,j]$.

Proof. ($\Rightarrow$) If s is a left seed of u then there exists a prefix p of s of length at least $n - j$ which is a suffix of u (see Fig. 2). We use here the fact, that $u[1\,..\,j]$ is the *longest* prefix of u covered by s. Hence, p is a border of u, and consequently $\mathsf{border}(u) \geq |p| \geq n - j$. Thus we obtain the desired inequality $j \geq \mathsf{per}(u)$.

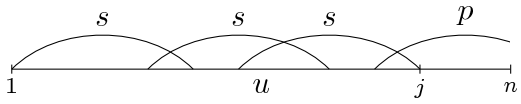

Fig. 2. Illustration of part ($\Rightarrow$) of Lemma 4.

($\Leftarrow$) The inequality $j \geq \mathsf{per}(u)$ implies that $v = u[1\,..\,j]$ is a left seed of u (see Fig. 3). Hence, by Observation 1b, the word s, which is a cover of v, is also a left seed of u.

Finally, the "in particular" part is a consequence of Observation 1, parts b and f. □

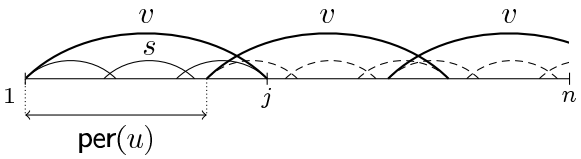

Fig. 3. Illustration of part (⇐) of Lemma 4.

Lemma 5. *Let $u \in \Sigma^n$ and let $\mathsf{C}[1\,..\,n]$ be its cover array. Then:*

$$\mathit{lseed}(u) = \min\{\,\mathsf{C}[j] \;:\; j \geq \mathit{per}(u)\}. \tag{1}$$

Proof. By Lemma 4, the length of the shortest left seed of u can be found among the values $\mathsf{C}[\mathsf{per}(u)], \ldots, \mathsf{C}[n]$. And conversely, for each of the values $\mathsf{C}[j]$ for $\mathsf{per}(u) \leq j \leq n$, there exists a left seed of u of length $\mathsf{C}[j]$. Thus $\mathsf{lseed}(u)$ equals the minimum of these values, which yields the formula (1). □

Clearly, the formula (1) provides an $O(n)$ time algorithm for computing the shortest left seed of the whole string u. We show that, employing some algorithmic techniques, one can use this formula to compute shortest left seeds for all prefixes of u, i.e., computing the left-seed array of u, also in $O(n)$ time.

Theorem 1. *For $u \in \Sigma^n$, its left-seed array can be computed in $O(n)$ time.*

Proof. Applying (1) to all prefixes of u, we obtain:

$$\mathsf{LSeed}[i] = \min\{\mathsf{C}[j] \;:\; \mathsf{P}[i] \leq j \leq i\}. \tag{2}$$

Recall that both the period array $\mathsf{P}[1\,..\,n]$ and the cover array $\mathsf{C}[1\,..\,n]$ of u can be computed in $O(n)$ time [1, 4, 6, 8].

The minimum in the formula (2) could be computed by data structures for Range-Minimum-Queries [9, 15], however in this particular case we can apply a much simpler algorithm. Note that $\mathsf{P}[i-1] \leq \mathsf{P}[i]$, therefore the intervals of the form $[\mathsf{P}[i], i]$ behave like a sliding window, i.e., both their endpoints are non-decreasing. We use a bidirectional queue Q which stores left-minimal elements in the current interval $[\mathsf{P}[i], i]$ (w.r.t. the value $\mathsf{C}[j]$). In other words, elements of Q are increasing and if Q during the step i contains an element j then $j \in [\mathsf{P}[i], i]$ and $\mathsf{C}[j] < \mathsf{C}[j']$ for all $j < j' \leq i$. We obtain an $O(n)$ time algorithm ComputeLeftSeedArray. □

ALGORITHM ComputeLeftSeedArray(u)

```
1: P[1..n] := period array of u;  C[1..n] := cover array of u;
2: Q := emptyBidirectionalQueue;
3: for i := 1 to n do
4:     while (not empty(Q)) and (front(Q) < P[i]) do popFront(Q);
5:     while (not empty(Q)) and (C[back(Q)] ≥ C[i]) do popBack(Q);
6:     pushBack(Q, i);
7:     LSeed[i] := C[front(Q)];
8:     { Q stores left-minimal elements of the interval [P[i], i] }
9: return  LSeed[1..n];
```

Now we proceed to an alternative algorithm computing the left-seed array, which also utilizes the criterion from Lemma 4. We start with an auxiliary algorithm ComputeR-Array. It computes an array $\mathsf{R}[1\,..\,n]$ which stores, as $\mathsf{R}[i]$, the length of the longest prefix of u for which $u[1\,..\,i]$ is the shortest cover, 0 if none.

ALGORITHM ComputeR-Array(u)

```
1: C[1..n] := cover array of u;
2: for i := 1 to n do R[i] := 0;
3: for i := 1 to n do R[C[i]] := i;
4: return  R[1..n];
```

The algorithm Alternative-ComputeLeftSeedArray computes the array LSeed from left to right. The current value of $\mathsf{LSeed}[i]$ is stored in the variable ls, note that this value never decreases (by Observation 1e). Equivalently, for each i we have $\mathsf{LSeed}[i-1] \le \mathsf{LSeed}[i] \le i$.

The particular value of $\mathsf{LSeed}[i]$ is obtained using the necessary and sufficient condition from Lemma 4: $\mathsf{LSeed}[i] = ls$ if ls is the smallest number such that $|w| \ge \mathsf{per}(u[1\,..\,i]) = \mathsf{P}[i]$, where w is the longest prefix of $u[1\,..\,i]$ that is covered by $u[1\,..\,ls]$. We slightly modify this condition, substituting w with the longest prefix w' of the very word u that is covered by $u[1\,..\,ls]$. Thus we obtain the condition $\mathsf{R}[ls] \ge \mathsf{P}[i]$ utilized in the pseudocode below.

ALGORITHM Alternative-ComputeLeftSeedArray(u)

```
1: P[1..n] := period array of u;  R[1..n] := ComputeR-Array(u);
2: LSeed[0] := 0;  ls := 0;
3: for i := 1 to n do
4:     { An invariant of the loop: ls = LSeed[i − 1]. }
5:     while R[ls] < P[i] do ls := ls + 1;
6:     LSeed[i] := ls;
7: return  LSeed[1..n];
```

Theorem 2. *Algorithm Alternative-ComputeLeftSeedArray runs in linear time.*

Proof. Recall that the arrays $\mathsf{P}[1\,..\,n]$ and $\mathsf{C}[1\,..\,n]$ can be computed in linear time [1, 4, 6, 8]. The array $\mathsf{R}[1\,..\,n]$ is obviously also computed in linear time.

It suffices to prove that the total number of steps of the while-loop in the algorithm Alternative-ComputeLeftSeedArray is linear in terms of n. In each step of the loop, the value of *ls* increases by one; this variable never decreases and it cannot exceed n. Hence, the while-loop performs at most n steps and the whole algorithm runs in $O(n)$ time. □

Concluding this section, we describe a linear-time algorithm computing the longest left-seed array, $\mathsf{LSeed}^M[1\,..\,n]$, of the string $u \in \Sigma^n$. The following lemma gives a simple characterization of the length of the longest left seed of the whole string u.

Lemma 6. *Let $u \in \Sigma^n$. If $\mathsf{per}(u) < n$ then $\mathsf{lseedmax}(u) = n - 1$, otherwise $\mathsf{lseedmax}(u) = 0$.*

Proof. First consider the case $\mathsf{per}(u) = n$. We show that $\mathsf{lseed}(u) = n$, consequently $\mathsf{lseedmax}(u)$ equals 0. Assume to the contrary that $\mathsf{lseed}(u) < n$. Then, a non-empty prefix of the minimal left seed of u, say w, is a suffix of u (consider the occurrence of the left seed that covers $u[n]$). Hence, $n - |w|$ is a period of u, a contradiction.

Assume now that $\mathsf{per}(u) < n$. Then u is a prefix of the word $u[1\,..\,\mathsf{per}(u)] \cdot u[1\,..\,n-1]$ which is covered by $u[1\,..\,n-1]$. Therefore $u[1\,..\,n-1]$ is a left seed of u, $\mathsf{lseedmax}(u) \geq n - 1$, consequently $\mathsf{lseedmax}(u) = n - 1$. □

Using Lemma 6 we obtain $\mathsf{LSeed}^M[i] = i - 1$ or $\mathsf{LSeed}^M[i] = 0$ for every i, depending on whether $\mathsf{P}[i] < i$ or not. We obtain the following result.

Theorem 3. *Longest left-seed array of $u \in \Sigma^n$ can be computed in $O(n)$ time.*

3 Computing Seeds of Given Length and Seed Array

In this section we show an $O(n^2)$ time algorithm computing the seed array $\mathsf{Seed}[1\,..\,n]$ of a given string $u \in \Sigma^n$, note that a trivial approach — computing the shortest seed for every prefix of u — yields $O(n^2 \log n)$ time complexity. In our solution we utilize a subroutine: testing whether u has a seed of a given length k. The following theorem shows that this test can be performed in $O(n)$ time.

Theorem 4. *It can be checked whether a given string $u \in \Sigma^n$ has a seed of a given length k in $O(n)$ time.*

Proof. Assume we have already computed in $O(n)$ time the suffix array SUF and the LCP array of longest common prefixes, see [6]. In the algorithm we start by dividing all factors of u of length k into groups corresponding to equal words. Every such group can be described as a maximal interval $[i\,..\,j]$ in the suffix array SUF, such that each of the values $\mathsf{LCP}[i+1], \mathsf{LCP}[i+2], \ldots, \mathsf{LCP}[j]$ is at least

k. The collection of such intervals can be constructed in $O(n)$ time by a single traversal of the LCP and SUF arrays (lines 1–9 of Algorithm SeedsOfAGivenLength). Moreover, using Bucket Sort, we can transform this representation into a collection of lists, each of which describes the set $Occ(v, u)$ for some factor v of u, $v \in \Sigma^k$ (lines 10–11 of the algorithm). This can be done in linear time, provided that we use the same set of buckets in each sorting and initialize them just once.

Now we process each of the lists separately, checking the conditions from Observation 3: in lines 14–18 of the algorithm we check the "maxgap" condition, and in line 19 the "border seed" condition, employing Fact 2.

Thus, having computed the arrays SUF and LCP, and the period arrays $\mathsf{P}[1\,..\,n]$ and $\mathsf{P}'[1\,..\,n]$ of u, we can find all seeds of u of length k in $O(n)$ total time. □

ALGORITHM SeedsOfAGivenLength(u, k)

```
 1: P[1..n] := period array of u;  P'[1..n] := suffix period array of u;
 2: SUF[1..n] := suffix array of u;  LCP[1..n] := lcp array of u;
 3: Lists := emptyList;
 4: j := 1;
 5: while j ≤ n do
 6:    List := {SUF[j]};
 7:    while j < n and LCP[j + 1] ≥ k do
 8:       j := j + 1;  List := append(List, SUF[j]);
 9:    j := j + 1;  Lists := append(Lists, List);
10: for all List in Lists do
11:    BucketSort(List); { using the same set of buckets }
12: for all List in Lists do
13:    first := prev := n;  last := 1;  covers := true;
14:    for all i in List do
15:       first := min(first, i);  last := max(last, i);
16:       if i > prev + k then
17:          covers := false;
18:       prev := i;
19:    if covers and (k ≥ max(P[first + k − 1], P'[last])) then
20:       print "u[first..(first + k − 1)] is a seed of u";
```

We compute the elements of the seed array $\mathsf{Seed}[1\,..\,n]$ from left to right, i.e., in the order of increasing lengths of prefixes of u. Note that $\mathsf{Seed}[i+1] \geq \mathsf{Seed}[i]$ for any $1 \leq i \leq n-1$, this is due to Observation 1d. If $\mathsf{Seed}[i+1] > \mathsf{Seed}[i]$ then we increase the current length of the seed by one letter at a time, in total at most $n-1$ such operations are performed. Each time we query for existence of a seed of a given length using the algorithm from Theorem 4. Thus we obtain $O(n^2)$ time complexity.

Theorem 5. *The seed array of a string $u \in \Sigma^n$ can be computed in $O(n^2)$ time.*

4 Alternative Algorithm for Shortest Seeds

In this section we present a new approach to shortest seeds computation based on very simple independent processing of disjoint chains in the suffix tree. It simplifies the computation of shortest seeds considerably.

Our algorithm is also based on a slightly modified version of Observation 3, formulated below as Lemma 7, which allows to relax the definition of maxgaps. We discuss an algorithmically easier version of maxgaps, called prefix maxgaps, and show that it can substitute maxgap values when looking for the shortest seed.

We start by analyzing the "border seed" condition. We introduce somewhat more abstract representation of sets of factors of u, called *prefix families*, and show how to find in them the shortest border seeds of u. Afterwards the key algorithm for computing prefix maxgaps is presented. Finally, both techniques are utilized to compute the shortest seed.

Let us fix the input string $u \in \Sigma^n$. For $v \in \Sigma^*$, by $PREF(v)$ we denote the set of all prefixes of v and by $PREF(v,k)$ we denote $PREF(v) \cap \Sigma^k\Sigma^*$ (*limited prefix subset*).

Let $\mathcal{F}$ be a family of limited prefix subsets of some factors of u, we call $\mathcal{F}$ a *prefix family*. Every element $PREF(v,k) \in \mathcal{F}$ can be represented in a canonical form, by a tuple of integers: $(first(v), last(v), k, |v|)$. Such a representation requires only constant space per element. By $\mathsf{bseed}(u,\mathcal{F})$ we denote the shortest border seed of u contained in some element of $\mathcal{F}$.

Example 1. Let $u = \mathtt{aabaababaabaaba}$ be the example word from Fig. 1. Let:

$$\mathcal{F} = \{PREF(\mathtt{abaab}, 4),\ PREF(\mathtt{babaa}, 4)\} = \{(2, 10, 4, 5),\ (6, 6, 4, 5)\}.$$

Note that $\bigcup \mathcal{F} = \{\mathtt{abaa}, \mathtt{abaab}, \mathtt{baba}, \mathtt{babaa}\}$. Then $\mathsf{bseed}(u,\mathcal{F}) = \mathtt{abaa}$.

The proof of the following fact is present implicitly in [11] (type-A and type-B seeds).

Theorem 6. *Let $u \in \Sigma^n$ and let $\mathcal{F}$ be a prefix family given in a canonical form. Then $\mathsf{bseed}(u,\mathcal{F})$ can be computed in linear time.*

Alternative proof of Theorem 6. There is an alternative algorithm for computing $\mathsf{bseed}(u,\mathcal{F})$, based on a special version of Find-Union data structure. Recall that $\mathsf{B}[1\,..\,n]$ is the border-array of u. Denote by $FirstGE(\mathcal{I}, c)$ (*first-greater-equal*) a query:

$$FirstGE(\mathcal{I}, c) \ = \ \min\{i \ : \ i \in \mathcal{I},\ \mathsf{B}[i] \geq c\},$$

where $\mathcal{I}$ is a subinterval of $[1\,..\,n]$. We assume that $\min\emptyset = +\infty$. A sequence of linear number of such queries, sorted according to non-decreasing values of c, can be easily answered in linear time, using an interval version of Find-Union data structure, see [7, 10]. The following algorithm applies the condition for border seed from Fact 2 to every element of $\mathcal{F}$, with $\mathsf{P}[first(s) + |s| - 1]$ substituted by $first(s) + |s| - 1 - \mathsf{B}[first(s) + |s| - 1]$. We omit the details. □

ALGORITHM ComputeBorderSeed(u, $\mathcal{F}$)

1: $bseed := +\infty$;
2: **for all** $(first(v), last(v), k, |v|)$ **in** $\mathcal{F}$, in non-decreasing order of $first(v)$ **do**
3: $\quad k := \max(\mathsf{P}'[last(v)],\ k)$;
4: $\quad \mathcal{I} := [first(v) + k - 1,\ first(v) + |v| - 1]$;
5: $\quad pos := FirstGE(\mathcal{I},\ first(v) - 1)$;
6: $\quad bseed := \min(bseed,\ pos - first(v) + 1)$;
7: **return** $bseed$;

Computation of the shortest seeds via prefix maxgaps. Let $T(u)$ be the suffix tree of u, recall that it can be constructed in $O(n)$ time [6, 8]. By $Nodes(u)$ we denote the set of factors of u corresponding to explicit nodes of $T(u)$, for simplicity we identify the nodes with the strings they represent. For $v \in Nodes(u)$, the set $Occ(v, u)$ corresponds to leaf list of the node v (i.e., the set of values of leaves in the subtree rooted at v), denoted as $LL(v)$. Note that $first(v) = \min LL(v)$ and $last(v) = \max LL(v)$, and such values can be computed for all $v \in Nodes(u)$ in $O(n)$ time. For $v \in Nodes(u)$, we define the *prefix maxgap* of v as:

$$\Delta(v) \ = \ \max\{\mathsf{maxgap}(w) \ : \ w \in PREF(v)\}.$$

Equivalently, $\Delta(v)$ is the maximum of maxgap values on the path from v to the root of $T(u)$. We introduce an auxiliary problem:

Prefix Maxgap Problem:
given a word $u \in \Sigma^n$, compute $\Delta(v)$ for all $v \in Nodes(u)$.

The following lemma (an alternative formulation of Observation 3) shows that prefix maxgaps can be used instead of maxgaps in searching for seeds. This is important since computation of prefix maxgaps $\Delta(v)$ is simple, in comparison with $\mathsf{maxgap}(v)$ — this is due to the fact that the $\Delta(v)$ values on each path down the suffix tree $T(u)$ are non-decreasing. Efficient computation of $\mathsf{maxgap}(v)$ requires using augmented height-balanced trees [5] or other rather sophisticated techniques [3]. The shortest-seed algorithm in [11] also computes prefix maxgaps instead of maxgaps, however this observation is missing in [11].

Lemma 7. *Let s be a factor of $u \in \Sigma^*$ and let w be the shortest element of $Nodes(u)$ such that $s \in PREF(w)$. The word s is a seed of u if and only if $|s| \geq \Delta(w)$ and s is a border seed of u.*

Proof. If s corresponds to an element of $Nodes(u)$, then $s = w$. Otherwise, s corresponds to an implicit node in an edge in the suffix tree, and w is the lower end of the edge. Note that in both cases we have $\Delta(w) \geq \mathsf{maxgap}(w) = \mathsf{maxgap}(s)$. By Observation 3, this implies part ($\Leftarrow$) of the conclusion. As for the part ($\Rightarrow$), it suffices to show that $|s| \geq \Delta(w)$.

Assume, to the contrary, that $|s| < \Delta(w)$. Let $v \in PREF(w) \cap Nodes(u)$ be the word for which $\mathsf{maxgap}(v) = \Delta(w)$, and let a, b be consecutive elements of the set $Occ(v, u)$ for which $a + \mathsf{maxgap}(v) = b$.

Let us note that no occurrence of s starts at any of the positions $a+1, \ldots, b-1$. Moreover, none of the suffixes of the form $u[i\,..\,n]$, for $a + 1 \leq i \leq b - 1$, is a prefix of s. Indeed, v is a prefix of s of length at most $n - b + 1$, and such an occurrence of s (or its prefix) would imply an extra occurrence of v. Note that at most $|s| \leq b - a - 1$ first positions in the interval $[a, b]$ can be covered by an occurrence of s in u (at position a or earlier) or by a suffix of s which is a prefix of u. Hence, position $b - 1$ is not covered by s at all, a contradiction. □

By Lemma 7, to complete the shortest seed algorithm it suffices to solve the Prefix Maxgap Problem (this is further clarified in the ComputeShortestSeed algorithm below). For this, we consider the following problem. By $SORT(X)$ we denote the sorted sequence of elements of $X \subseteq \{1, 2, \ldots, n\}$.

Chain Prefix Maxgap Problem

Input: a family of disjoint sets $X_1, X_2, \ldots, X_k \subseteq \{1, 2, \ldots, n\}$ together with $SORT(X_1 \cup X_2 \cup \ldots \cup X_k)$. The size of the input is $m = \sum |X_i|$.

Output: the numbers $\Delta_i = \max_{j \leq i} \mathsf{maxgap}(X_j \cup X_{j+1} \cup \ldots \cup X_k)$.

Theorem 7. *The Chain Prefix Maxgap Problem can be solved in $O(m)$ time using an auxiliary array of size n.*

Proof. Initially we have the list $L = SORT(X_1 \cup X_2 \cup \ldots \cup X_k)$. Let *pred* and *suc* denote the predecessor and successor of an element of L. The elements of L store a Boolean flag *marked*, initially set to false. In the algorithm we use an auxiliary array $pos[1\,..\,n]$ such that $pos[i]$ is a pointer to the element of value i in L, if there is no such element then the value of $pos[i]$ can be arbitrary. Obviously the algorithm takes $O(m)$ time. □

ALGORITHM ChainPrefixMaxgap(L)

```
1: Δ_1 := maxgap(L); { naive computation }
2: for j := 2 to k do
3:     Δ_j := Δ_{j-1};
4:     for all i in X_{j-1} do marked(pos[i]) := true;
5:     for all i in X_{j-1} do
6:         p := pred(pos[i]);  q := suc(pos[i]);
7:         if (p ≠ nil) and (q ≠ nil) and (not marked(p))
           and (not marked(q)) then
8:             Δ_j := max(Δ_j, value(q) − value(p));
9:         delete(L, pos[i]);
```

Theorem 8. *The Prefix Maxgap Problem can be reduced to a collection of Chain Prefix Maxgap Problems of total size $O(n \log n)$.*

Proof. We solve a more abstract version of the Prefix Maxgap Problem. We are given an arbitrary tree T with n leaves annotated with distinct integers from the interval $[1, n]$, and we need to compute the values $\Delta(v)$ for all $v \in \mathit{Nodes}(T)$, defined as follows: $\mathsf{maxgap}(v) = \mathsf{maxgap}(LL(v))$, where $LL(v)$ is the leaf list of v, and $\Delta(v)$ is the maximum of the values maxgap on the path from v to the root of T. We start by sorting $LL(\mathit{root}(T))$, which can be done in $O(n)$ time. Throughout the algorithm we store a global auxiliary array $\mathit{pos}[1\,..\,n]$, required in the ChainPrefixMaxgap algorithm.

Let us find a *heaviest path* P in T, i.e., a path from the root down to a leaf, such that all *hanging* subtrees are of size at most $|T|/2$ each. The values of $\Delta(v)$ for $v \in P$ can all be computed in $O(n)$ time, using a reduction to the Chain Prefix Maxgap Problem (see Fig. 4).

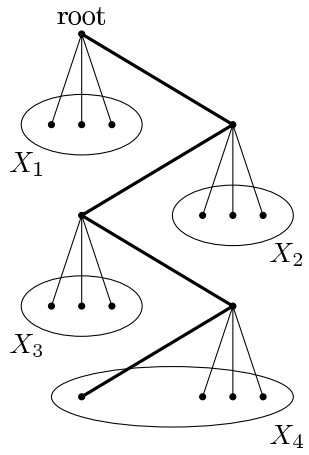

Fig. 4. A tree with an example heaviest path P (in bold). The values $\Delta(v)$ for $v \in P$ can be computed using a reduction to the Chain Prefix Maxgap Problem with the sets X_1 through X_4.

Then we perform the computation recursively for the hanging subtrees, previously sorting $LL(T')$ for each hanging subtree T'. Such sorting operations can be performed in $O(n)$ total time for all hanging subtrees.

At each level of recursion we need a linear amount of time, and the depth of recursion is logarithmic. Hence, the total size of invoked Chain Prefix Maxgap Problems is $O(n \log n)$. □

Now we proceed to the shortest seed computation. In the algorithm we consider all factors of u, dividing them into groups corresponding to elements of $\mathit{Nodes}(u)$. Let $w \in \mathit{Nodes}(u)$ and let v be its parent. Let $s \in \mathit{PREF}(w)$ be a word containing v as a proper prefix, i.e., $s \in \mathit{PREF}(w, |v| + 1)$. By Lemma 7, the word s is a seed of u if and only if $|s| \geq \Delta(w)$ and s is a border seed of u.

Using the previously described reductions (Theorems 6–8), we obtain the following algorithm:

ALGORITHM ComputeShortestSeed(u)

1: Construct the suffix tree $T(u)$ for the input string u;
2: Solve the Prefix Maxgap Problem for $T(u)$ using the ChainPrefixMaxgap
3: algorithm — in $O(n \log n)$ total time (Theorems 7 and 8);
4: $\mathcal{F} := \{\ PREF(w,\ \max(|v|+1, \Delta(w)))\ :\ (v,w)$ is an edge in $T(u)\ \}$;
5: **return** $\mathsf{bseed}(u, \mathcal{F})$; { Theorem 6 }

Observe that the *workhorse* of the algorithm is the chain version of the Prefix Maxgap Problem, which has a fairly simple linear time solution. The main problem is of a structural nature, we have a collection of very simple problems each computable in linear time but the total size is not linear. This identifies the bottleneck of the algorithm from the complexity point of view.

5 Long Seeds

Note that the most time-expensive part of the ComputeShortestSeed algorithm is the computation of prefix maxgaps, all the remaining operations are performed in $O(n)$ time. Using this observation we can show a more efficient algorithm computing the shortest seed provided that its length m is sufficiently large. For example if $m = \Theta(n)$ then we obtain an $O(n)$ time algorithm for the shortest seed.

Theorem 9. *One can check if the shortest seed of a given string u has length at least m in $O(n \log(n/m))$ time, where $n = |u|$. If so, a corresponding seed can be reported within the same time complexity.*

Proof. We show how to modify the ComputeShortestSeed algorithm. Denote by s the shortest seed of u, $|s| = m$.

By Observation 1g, the longest overlap between consecutive occurrences of s in u is at most $\frac{m}{2}$, therefore the number of occurrences of s in u is at most $\frac{2n}{m}$. Hence, searching for the shortest seed of length at least m, it suffices to consider nodes v of the suffix tree $T(u)$ for which: $|v| \geq m$ and $|LL(v)| \leq \frac{2n}{m}$.

Thus, we are only interested in prefix maxgaps for nodes in several subtrees of $T(u)$, each of which contains $O(n/m)$ nodes. Thanks to the small size of each subtree, the algorithm ComputeShortestSeed finds all such prefix maxgaps in $O(n \log(n/m))$ time. Please note that using this algorithm for each node we obtain a prefix maxgap only in its subtree (not necessarily in the whole tree), however Lemma 7 can be simply adjusted to such a modified definition of prefix maxgaps. □

References

1. A. Apostolico and D. Breslauer. Of periods, quasiperiods, repetitions and covers. In *Structures in Logic and Computer Science*, pages 236–248, 1997.
2. A. Apostolico and A. Ehrenfeucht. Efficient detection of quasiperiodicities in strings. *Theor. Comput. Sci.*, 119(2):247–265, 1993.
3. O. Berkman, C. S. Iliopoulos, and K. Park. The subtree max gap problem with application to parallel string covering. *Inf. Comput.*, 123(1):127–137, 1995.
4. D. Breslauer. An on-line string superprimitivity test. *Inf. Process. Lett.*, 44(6):345–347, 1992.
5. G. S. Brodal and C. N. S. Pedersen. Finding maximal quasiperiodicities in strings. In R. Giancarlo and D. Sankoff, editors, *CPM*, volume 1848 of *Lecture Notes in Computer Science*, pages 397–411. Springer, 2000.
6. M. Crochemore, C. Hancart, and T. Lecroq. *Algorithms on Strings*. Cambridge University Press, 2007.
7. M. Crochemore, C. S. Iliopoulos, M. Kubica, J. Radoszewski, W. Rytter, and T. Walen. Extracting powers and periods in a string from its runs structure. In E. Chávez and S. Lonardi, editors, *SPIRE*, volume 6393 of *Lecture Notes in Computer Science*, pages 258–269. Springer, 2010.
8. M. Crochemore and W. Rytter. *Jewels of Stringology*. World Scientific, 2003.
9. J. Fischer and V. Heun. A new succinct representation of RMQ-information and improvements in the enhanced suffix array. In B. Chen, M. Paterson, and G. Zhang, editors, *ESCAPE*, volume 4614 of *Lecture Notes in Computer Science*, pages 459–470. Springer, 2007.
10. H. N. Gabow and R. E. Tarjan. A linear-time algorithm for a special case of disjoint set union. *Proceedings of the 15th Annual ACM Symposium on Theory of Computing (STOC)*, pages 246–251, 1983.
11. C. S. Iliopoulos, D. W. G. Moore, and K. Park. Covering a string. *Algorithmica*, 16(3):288–297, 1996.
12. Y. Li and W. F. Smyth. Computing the cover array in linear time. *Algorithmica*, 32(1):95–106, 2002.
13. M. Lothaire, editor. *Algebraic Combinatorics on Words*. Cambridge University Press, 2001.
14. M. Lothaire, editor. *Applied Combinatorics on Words*. Cambridge University Press, 2005.
15. K. Sadakane. Succinct data structures for flexible text retrieval systems. *J. Discrete Algorithms*, 5(1):12–22, 2007.